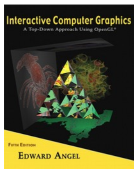

Interactive Computer Graphics
A Top-Down Approach Using OpenGL
Fifth Edition
Edward Angel

Name: ______________________ Term # ____

Homework 4 SOLUTIONS

(400 points)

NOTE: **Chapter 4 of the textbook shows the shows the geometric objects and transformations.**

Part A is intended to be done by hand.

Part B is an openGL application.

A. (300 pts) Paper and Pencil

***(Guidelines:** Read the material from the textbook chapter, you can use textbook figures to exemplify your answer, use keywords, summarize your answer, but the answer **cannot be longer the 7 lines!**)*

4.1 SCALARS, POINTS, AND VECTORS

a. Describe how scalars, points, and vectors are used in describing geometric objects like lines, polygons, and polyhedra.
ANSWER:
(Angel pp. 160)

In computer graphics, we work with sets of geometric objects, such as lines, polygons, and polyhedra. Such objects exist in a three- dimensional world and have properties that can be described using concepts such as length and angles. As we discovered working in two dimensions, we can define most geometric objects using a limited set of simple entities. These basic geometric objects and the relationships among them can be described using three fundamental types: scalars, points, and vectors. Although we will consider each type from a geometric perspective, each of these types also can be defined formally, as in Appendix B, as obeying a set of axioms. Although ultimately we will use the geometric instantiation of each type, we want to take great care in distinguishing between the abstract definition of each entity and any particular example, or implementation, of it. By taking care here, we can avoid many subtle pitfalls later. Although we will work in three- dimensional spaces, virtually all our results will hold in n- dimensional spaces.

4.1.1 Geometric Objects

a. Define point.
ANSWER:
(Angel pp. 160)
Our fundamental geometric object is a point. In a three- dimensional geometric sys-tem, a point is a location in space. The only property that a point possesses is that point’s location; a mathematical point has neither a size nor a shape.

b. Define scalar.
ANSWER:
(Angel pp. 160)
Points are useful in specifying geometric objects but are not sufficient by themselves. We need real numbers to specify quantities such as the distance between two points. Real numbers and complex numbers, which we will use occasionally, are examples of scalars. Scalars are objects that obey a set of rules that are abstractions of the operations of ordinary arithmetic. Thus, addition and multiplication are defined and obey the usual rules such as commutivity and associativity. Every scalar has multiplicative and additive inverses, which implicitly define subtraction and division.

c. Define vector.
ANSWER:
(Angel pp. 160)
In computer graphics, we often connect points with directed line segments, as shown in

Figure 4.1.

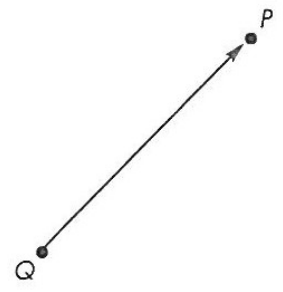


FIGURE 4.1 Directed line segment that connects points.

A directed line segment has both magnitude its length and direction its orientation and thus is a vector. Because vectors have no fixed position, the directed line segments shown in Figure 4.2 are identical because they have the same direction and magnitude.
We will often use the terms vector and directed line segment synonymously. Vectors can have their lengths altered by real numbers. Thus, in Figure 4.3(a), line segment A has the same direction as line segment B, but B has twice the length that A has, so we can write B = 2A.

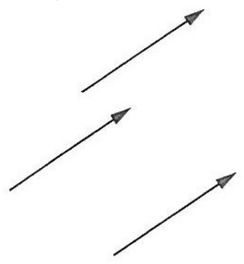


FIGURE 4.2 Identical vectors.

We can also combine directed line segments by the head- to- tail rule, as shown in Figure 4.3(b). Here, we connect the head of vector A to the tail of vector C to form a new vector D, whose magnitude and direction are determined by the line segment from the tail of A to the head of C. We call this new vector, D, the sum of A and C and write D = A + C. Because vectors have no fixed positions, we can move any two vectors as necessary to form their sum graphically. Note that we have described two fundamental operations: the addition of two vectors and the multiplication of a vector by a scalar.

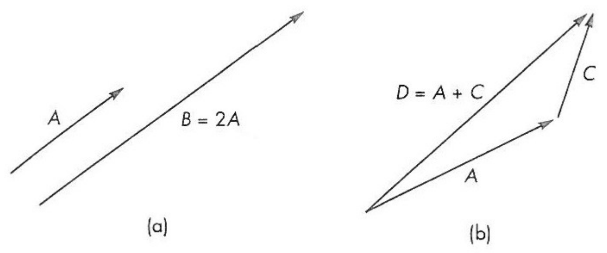


FIGURE 4.3 (a) Parallel line segments. (b) Addition of line segments.

d. Define zero vector.
ANSWER:

(Angel pp. 161)
If we consider two directed line segments, A and E, as shown in Figure 4.4, with the same length but opposite directions, their sum as defined by the head- to- tail addition has no length. This sum forms a special vector called the zero vector, which we denote 0, that has a magnitude of zero. Because it has no length, the orientation of this vector is undefined. We say that E is the inverse of A and we can write E =- A. Using inverses of vectors, scalar- vector expressions such as A + 2B - 3C make sense.

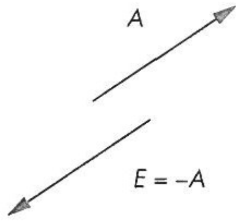


FIGURE 4.4 Inverse vectors.

e. Define valid point-point and point-scalar operations.
ANSWER:
(Angel pp. 161)
Although we can multiply a vector by a scalar to change its length, there are no obvious sensible operations between two points that produce another point. Nor are there operations between a point and a scalar that produce a point.

f. Define valid point-vector operations.
ANSWER:
(Angel pp. 161)
There is, however, an operation between points and directed line segments (vectors), as illustrated in Figure 4.5.
We can use a directed line segment to move from one point to another. We call this operation point-vector addition, and it produces a new point. We write this operation as P = Q + v. We can see that the vector v displaces the point Q to the new location P.

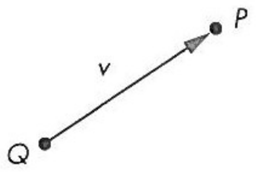


FIGURE 4.5 Point-vector addition.

g. Define point-point subtraction.
ANSWER:
(Angel pp. 161)
Looking at things slightly differently, any two points define a directed line segment or vector from one point to the second. We call this operation point- point subtraction, and

we can write it as v = P - Q.

h. Is the expression involving scalars, points, and vectors valid?

P + 3v	valid?	YES	NO
P + 3Q - v	valid?	YES	NO
2P - Q + 3v	valid?	YES	NO

ANSWER:
(Angel pp. 161)

P + 3v	valid?	YES	
P + 3Q - v	valid?	NO	no point-point addition defined
2P - Q + 3v	valid?	YES	

4.1.2 Coordinate-Free Geometry

a. What is a coordinate system allowing to answer?
ANSWER:
(Angel pp. 160)
Points exist in space regardless of any reference or coordinate system. Thus, we do not need a coordinate system to define a point or a vector. This fact may seem counter to your experiences, but it is crucial to understanding geometry and how to build graphics systems. Consider the two- dimensional example shown in Figure 4.6. Here we see a coordinate system defined by two axes, an origin, and a simple geometric object, a square. We can refer to the point at the lower- left corner of the square as having coordinates (1,1) and note that the sides of the square are orthogonal to each other and that the point at (3,1) is two units from the point at (1,1). Now suppose that we remove the axes as shown in Figure 4.7. We can no longer specify where the points are. But those locations were relative to an arbitrary location of the origin and the orientation of the axes. What is more important is that the fundamental geometric relationships are preserved. The square is still a square, orthogonal lines are still orthogonal, and distances between points remain the same. Of course, we may find it inconvenient, at best, to refer to a specific point as that point over there or the blue point to the right of the red one. Coordinate systems and frames (see Section 4.3) solve this reference problem, but for now we want to see just how far we can get following a coordinate- free approach that does not require an arbitrary reference system.

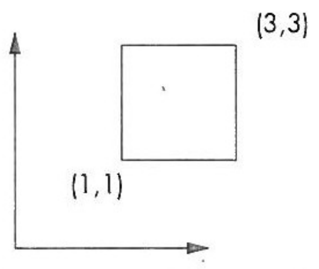


FIGURE 4.6 Object and coordinate system.

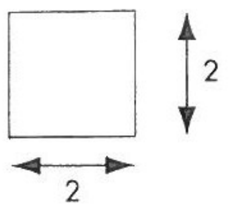


FIGURE 4.7 Object without coordinate system.

4.1.5 Geometric ADTs

a. What is the magnitude of a vector?
ANSWER:
(Angel pp. 164)
We have not as yet intro-duced any reference system, such as a coordinate system; thus, for vectors and points, this notation refers to the abstract objects, rather than to these objects representations in a particular reference system. We use boldface letters for the latter in Section 4.3. The magnitude of a vector v is a real number denoted by | v|.

$$|\alpha v| = |\alpha||v|,$$

and the direction of αv is the same as the direction of v if a is positive and the opposite direction if α is negative.

4.1.6 Lines

a. What is a parametric form of a line?
ANSWER:
(Angel pp. 165)
The sum of a point and a vector (or the subtraction of two points) leads to the notion of a line in an affine space. Consider all points of the form

$$P(\alpha) = P_0 + \alpha d,$$

where P0 is an arbitrary point, d is an arbitrary vector, and α is a scalar that can vary over some range of values. Given the rules for combining points, vectors, and scalars in an affine space, for any value of α, evaluation of the function P(α) yields a point. For geometric vectors (directed line segments), these points lie on a line, as shown in Figure 4.10. This form is known as the **parametric form of the line** because we generate points on the line by varying the parameter a. For α = 0, the line passes through the point P0, and as α is increased, all the points generated lie in the direction of the vector d. If we restrict α to nonnegative values, we get the ray emanating from P0 and going in the direction of d. Thus, a line is infinitely long in both directions, a line segment is a finite piece of a line between two points, and a ray is infinitely long in one direction.

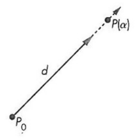


FIGURE 4.10 Line in an affine space.

4.1.7 Affine Sums

a. What is affine addition?
ANSWER:
(Angel pp. 165)
Whereas in an affine space the addition of two vectors, the multiplication of a vector by a scalar, and the addition of a vector and a point are defined, the addition of two arbitrary points and the multiplication of a point by a scalar are not. However, there is an operation called affine addition that has certain elements of these latter two operations. For any point Q, vector v, and positive scalar α,

$$P = Q + \alpha v$$

describes all points on the line from Q in the direction of v, as shown in Figure 4.11. However, we can always find a point R such that

$$v = R - Q;$$

thus

$$P = Q + \alpha(R - Q) = \alpha R + (1 - \alpha)Q.$$

This operation looks like the addition of two points and leads to the equivalent form

$$P = \alpha_1 R + \alpha_2 Q,$$

where

$$\alpha_1 + \alpha_2 = 1.$$

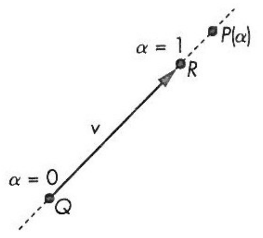
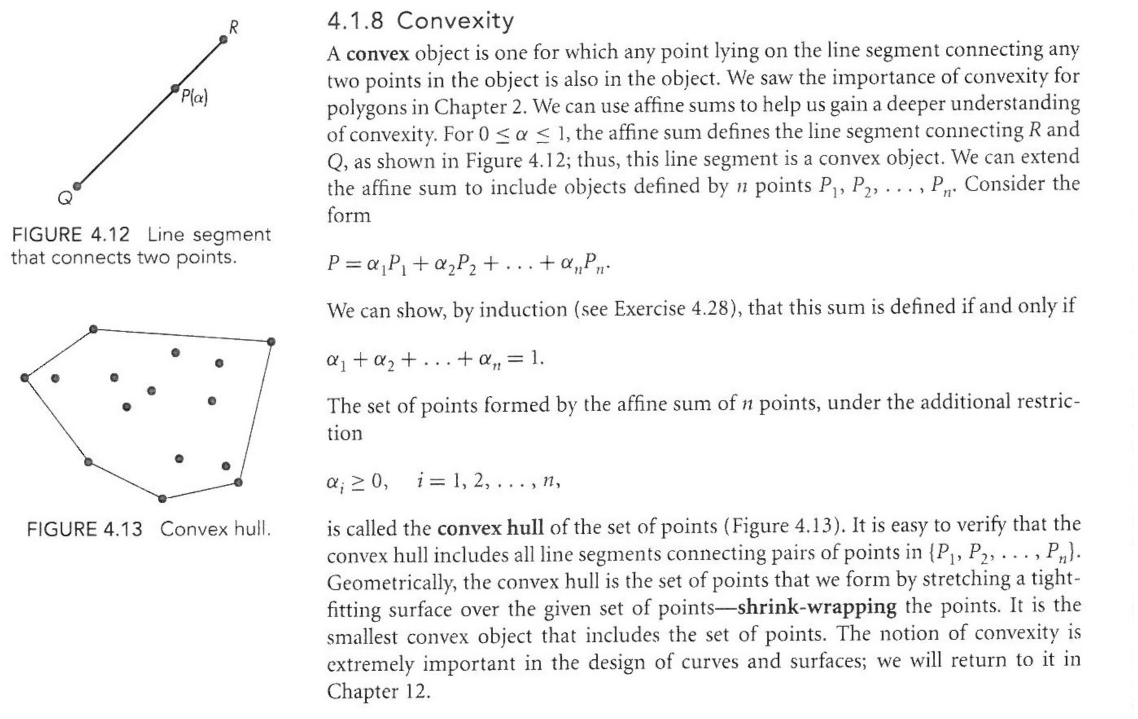


FIGURE 4.11 Affine addition.

4.1.8 Convexity

a. What is a convex object and convex hull?
ANSWER:
(Angel pp. 166)



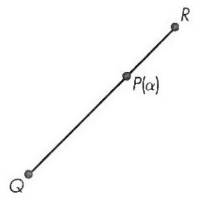


FIGURE 4.12 Line segment that connects two points.

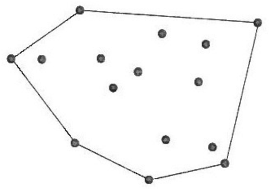


FIGURE 4.13 Convex hull.

4.1.8 Convexity

A **convex** object is one for which any point lying on the line segment connecting any two points in the object is also in the object. We saw the importance of convexity for polygons in Chapter 2. We can use affine sums to help us gain a deeper understanding of convexity. For $0 \leq \alpha \leq 1$, the affine sum defines the line segment connecting R and Q, as shown in Figure 4.12; thus, this line segment is a convex object. We can extend the affine sum to include objects defined by n points $P_1, P_2, \ldots, P_n$. Consider the form

$$P = \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n.$$

We can show, by induction (see Exercise 4.28), that this sum is defined if and only if

$$\alpha_1 + \alpha_2 + \ldots + \alpha_n = 1.$$

The set of points formed by the affine sum of n points, under the additional restriction

$$\alpha_i \geq 0, \quad i = 1, 2, \ldots, n,$$

is called the **convex hull** of the set of points (Figure 4.13). It is easy to verify that the convex hull includes all line segments connecting pairs of points in $\{P_1, P_2, \ldots, P_n\}$. Geometrically, the convex hull is the set of points that we form by stretching a tight-fitting surface over the given set of points—**shrink-wrapping** the points. It is the smallest convex object that includes the set of points. The notion of convexity is extremely important in the design of curves and surfaces; we will return to it in Chapter 12.

4.1.9 Dot Product and Cross Product

a. What is a dot product?
ANSWER:
(Angel pp. 166)

Many of the geometric concepts relating the orientation between two vectors are in terms of the **dot** (**inner**) and **cross** (**outer**) **products** of two vectors. The dot product of u and v is written $u \cdot v$ (see Appendix B). If $u \cdot v = 0$, u and v are said to be **orthogonal**. In a Euclidean space, the magnitude of a vector is defined. The square of the magnitude of a vector is given by the dot product

$$|u|^2 = u \cdot u.$$

The cosine of the angle between two vectors is given by

$$\cos\theta = \frac{u \cdot v}{|u||v|}.$$

In addition, $|u|\cos\theta = u \cdot v/|v|$ is the length of the orthogonal projection of u onto v, as shown in Figure 4.14. Thus, the dot product expresses the geometric result that the shortest distance from a point (the end of the vector u) to the line segment v is obtained by drawing the vector orthogonal to v from the end of u. We can also see that the vector u is composed of the vector sum of the orthogonal projection of u on v and a vector orthogonal to v.

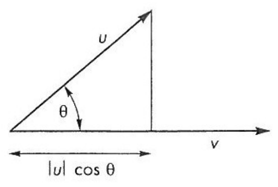


FIGURE 4.14 Dot product and projection.

a. What is a cross product?
ANSWER:
(Angel pp. 167)

In a vector space, a set of vectors is **linearly independent** if we cannot write one of the vectors in terms of the others using scalar-vector addition. A vector space has a **dimension**, which is the maximum number of linearly independent vectors that we can find. Given any three linearly independent vectors in a three-dimensional space, we can use the dot product to construct three vectors, each of which is orthogonal to the other two. This process is outlined in Appendix B. We can also use two non-parallel vectors, u and v, to determine a third vector n that is orthogonal to them (Figure 4.15). This vector is the **cross product**

$$n = u \times v.$$

Note that we can use the cross product to derive three mutually orthogonal vectors in a three-dimensional space from any two nonparallel vectors. Starting again with u and v, we first compute n as before. Then, we can compute w by

$$w = u \times n,$$

and u, n, and w are mutually orthogonal.

The cross product is derived in Appendix C, using the representation of the vectors that gives a direct method for computing it. The magnitude of the cross product gives the magnitude of the sine of the angle θ between u and v,

$$|\sin\theta| = \frac{|u \times v|}{|u||v|}.$$

Note that the vectors u, v, and n form a **right-handed coordinate system**; that is, if u points in the direction of the thumb of the right hand and v points in the direction of the index finger, then n points in the direction of the middle finger.

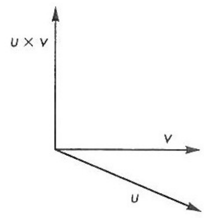


FIGURE 4.15 Cross product.

4.1.10 Planes

a. What is a parametric form of a plane?

ANSWER:

(Angel pp. 167)

A **plane** in an affine space can be defined as a direct extension of the parametric line. From simple geometry, we know that three points not on the same line determine a unique plane. Suppose that P, Q, and R are three such points in an affine space. The line segment that joins P and Q is the set of points of the form

$$S(\alpha) = \alpha P + (1 - \alpha)Q, \qquad 0 \le \alpha \le 1.$$

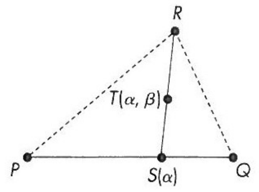


FIGURE 4.16 Formation of a plane.

Suppose that we take an arbitrary point on this line segment and form the line segment from this point to R, as shown in Figure 4.16. Using a second parameter β, we can describe points along this line segment as

$$T(\beta) = \beta S + (1 - \beta)R, \qquad 0 \le \beta \le 1.$$

Such points are determined by both α and β and form the plane determined by P, Q, and R. Combining the preceding two equations, we obtain one form of the equation of a plane:

$$T(\alpha, \beta) = \beta[\alpha P + (1 - \alpha)Q] + (1 - \beta)R.$$

We can rearrange this equation in the following form:

$$T(\alpha, \beta) = P + \beta(1 - \alpha)(Q - P) + (1 - \beta)(R - P).$$

Noting that $Q - P$ and $R - P$ are arbitrary vectors, we have shown that a plane can also be expressed in terms of a point, P_0, and two nonparallel vectors, u and v, as

$$T(\alpha, \beta) = P_0 + \alpha u + \beta v.$$

We can also observe that for $0 \le \alpha, \beta \le 1$, all the points $T(\alpha, \beta)$ lie in the triangle formed by P, Q, and R. If a point P lies in the plane, then

$$P - P_0 = \alpha u + \beta v.$$

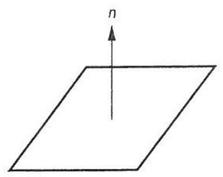


FIGURE 4.17 Normal to a plane.

We can find a vector w that is orthogonal to both u and v, as shown in Figure 4.17. If we use the cross product

$$n = u \times v,$$

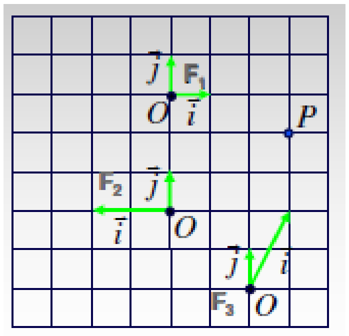
then the equation of the plane becomes

$$n \cdot (P - P_0) = 0.$$

The vector n is perpendicular, or orthogonal, to the plane; it is called the **normal** to the plane. The forms $P(\alpha)$, for the line, and $T(\alpha, \beta)$, for the plane, are known as **parametric forms** because they give the value of a point in space for each value of the parameters α and β.

4.3 COORDINATE SYSTEMS AND FRAMES

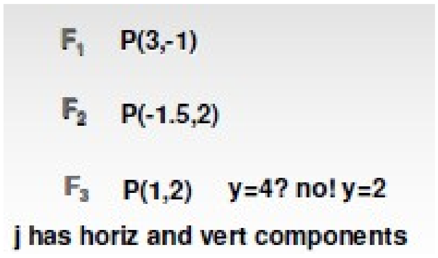
a. Consider the three frame systems below.



A point P can be described by:

$$P = O + x\vec{i} + y\vec{j}$$

Can you represent point P in the three frames, F_1, F_2, and F_3?
ANSWER:



F_1	P(3,-1)
F_2	P(-1.5,2)
F_3	P(1,2) y=4? no! y=2

j has horiz and vert components

<u>*4.9.4 CONCATENATION OF TRANSFORMATIONS*</u>

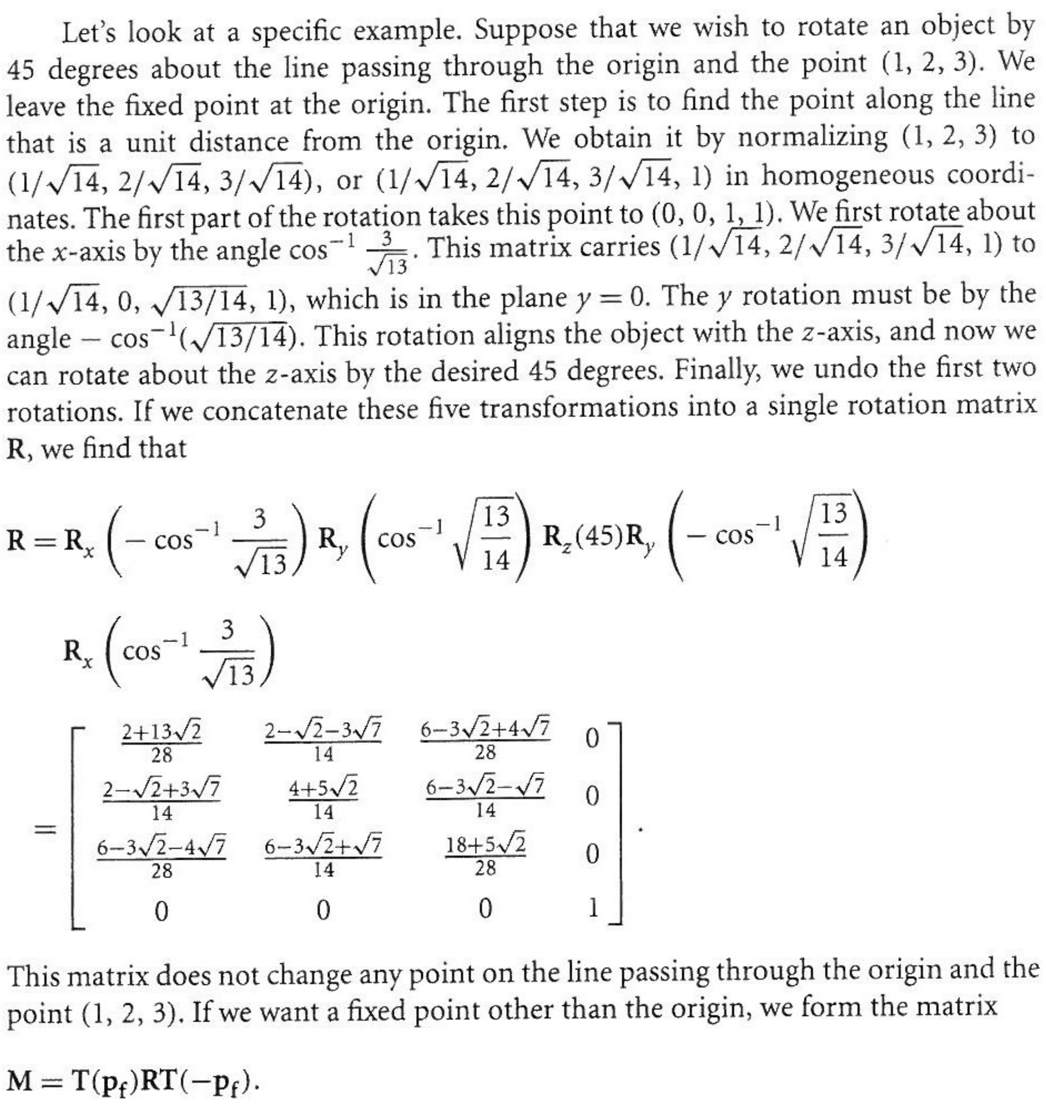
a. Suppose that we wish to rotate an object by 45 degrees about the line passing through the origin and the point (1, 2, 3). We leave the fixed point at the origin.

Use the rotation matrix that calculates rotation about an arbitrary axis on page 212:

$$\mathbf{R} = \mathbf{R}_x(-\theta_x)\mathbf{R}_y(-\theta_y)\mathbf{R}_z(\theta)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x).$$

$$\mathbf{M} = \mathbf{T}(\mathbf{p}_0)\mathbf{R}_x(-\theta_x)\mathbf{R}_y(-\theta_y)\mathbf{R}_z(\theta)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)\mathbf{T}(-\mathbf{p}_0).$$

And calculate M for the above rotation.
ANSWER:
(Angel pp. 214)



Let's look at a specific example. Suppose that we wish to rotate an object by 45 degrees about the line passing through the origin and the point (1, 2, 3). We leave the fixed point at the origin. The first step is to find the point along the line that is a unit distance from the origin. We obtain it by normalizing (1, 2, 3) to $(1/\sqrt{14}, 2/\sqrt{14}, 3/\sqrt{14})$, or $(1/\sqrt{14}, 2/\sqrt{14}, 3/\sqrt{14}, 1)$ in homogeneous coordinates. The first part of the rotation takes this point to (0, 0, 1, 1). We first rotate about the x-axis by the angle $\cos^{-1}\frac{3}{\sqrt{13}}$. This matrix carries $(1/\sqrt{14}, 2/\sqrt{14}, 3/\sqrt{14}, 1)$ to $(1/\sqrt{14}, 0, \sqrt{13/14}, 1)$, which is in the plane $y = 0$. The y rotation must be by the angle $-\cos^{-1}(\sqrt{13/14})$. This rotation aligns the object with the z-axis, and now we can rotate about the z-axis by the desired 45 degrees. Finally, we undo the first two rotations. If we concatenate these five transformations into a single rotation matrix $\mathbf{R}$, we find that

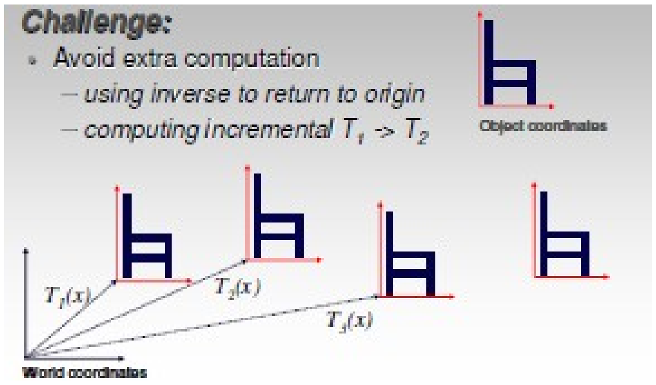
$$\mathbf{R} = \mathbf{R}_x\left(-\cos^{-1}\frac{3}{\sqrt{13}}\right)\mathbf{R}_y\left(\cos^{-1}\sqrt{\frac{13}{14}}\right)\mathbf{R}_z(45)\mathbf{R}_y\left(-\cos^{-1}\sqrt{\frac{13}{14}}\right)$$
$$\mathbf{R}_x\left(\cos^{-1}\frac{3}{\sqrt{13}}\right)$$

$$= \begin{bmatrix} \frac{2+13\sqrt{2}}{28} & \frac{2-\sqrt{2}-3\sqrt{7}}{14} & \frac{6-3\sqrt{2}+4\sqrt{7}}{28} & 0 \\ \frac{2-\sqrt{2}+3\sqrt{7}}{14} & \frac{4+5\sqrt{2}}{14} & \frac{6-3\sqrt{2}-\sqrt{7}}{14} & 0 \\ \frac{6-3\sqrt{2}-4\sqrt{7}}{28} & \frac{6-3\sqrt{2}+\sqrt{7}}{14} & \frac{18+5\sqrt{2}}{28} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This matrix does not change any point on the line passing through the origin and the point (1, 2, 3). If we want a fixed point other than the origin, we form the matrix

$$\mathbf{M} = \mathbf{T}(\mathbf{p}_f)\mathbf{R}\mathbf{T}(-\mathbf{p}_f).$$

4.10 OPENGL TRANSFORMATION MATRICES

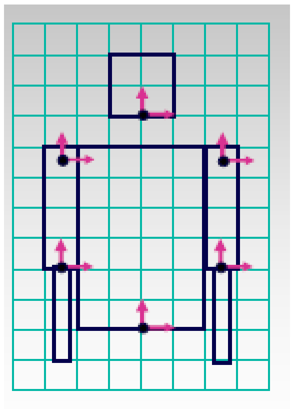


Challenge:

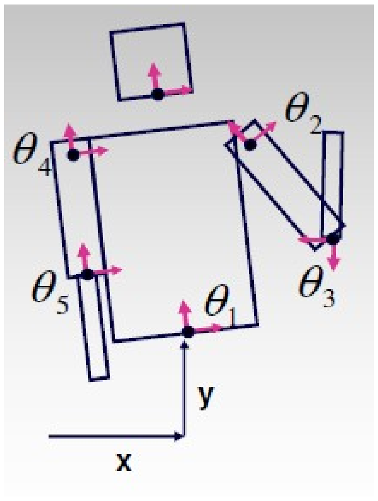
- Avoid extra computation
  - *using inverse to return to origin*
  - *computing incremental T_1 -> T_2*

Object coordinates
$T_1(x)$
$T_2(x)$
$T_3(x)$
World coordinates

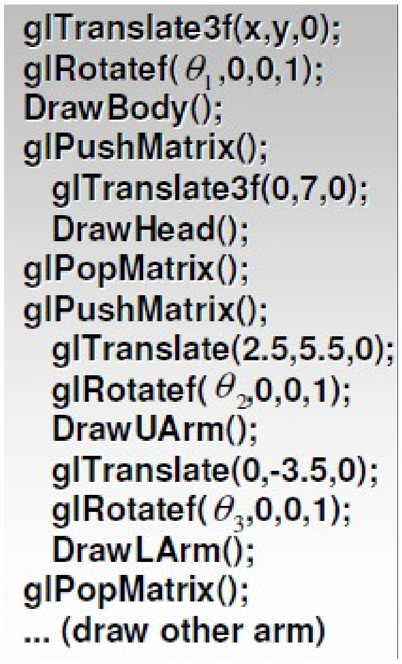
a. Consider the robot model below.



A movement of the robot model is described below:



Can you write the OPENGL function call that produce this move?
Assume that DrawBody(); will draw the body; Draw Head (); will draw the head; DrawUArm(); will draw the upper arm; DrawLArm(); will draw the lower arm.
(you need to use glPushMatrix(); glPopMatrix(); glTranslate(x,y,z); glRotate(θ1,θ2, θ3);
ANSWER:



```
glTranslate3f(x,y,0);
glRotatef($\theta_1$,0,0,1);
DrawBody();
glPushMatrix();
  glTranslate3f(0,7,0);
  DrawHead();
glPopMatrix();
glPushMatrix();
  glTranslate(2.5,5.5,0);
  glRotatef($\theta_2$,0,0,1);
  DrawUArm();
  glTranslate(0,-3.5,0);
  glRotatef($\theta_3$,0,0,1);
  DrawLArm();
glPopMatrix();
... (draw other arm)
```

B. (100 pts) Visual Studio 2008 C++ Project

B1. Create Visual Studio 2008 C++, Empty Project, Homework4:
Create a .c file **colorcube.c**
with the content:

colorcube.c

```
/* colorcube.c */

#include <stdlib.h>

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

        GLfloat vertices[] = {-1.0,-1.0,-1.0,1.0,-1.0,-1.0,
        1.0,1.0,-1.0, -1.0,1.0,-1.0, -1.0,-1.0,1.0,
        1.0,-1.0,1.0, 1.0,1.0,1.0, -1.0,1.0,1.0};

        GLfloat colors[] = {0.0,0.0,0.0,1.0,0.0,0.0,
        1.0,1.0,0.0, 0.0,1.0,0.0, 0.0,0.0,1.0,
        1.0,0.0,1.0, 1.0,1.0,1.0, 0.0,1.0,1.0};

   GLubyte cubeIndices[]={0,3,2,1,2,3,7,6,0,4,7,3,1,2,6,5,4,5,6,7,0,1,5,4};



static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 2;

void display(void)
{
   /* display callback, clear frame buffer and z buffer,
     rotate cube and draw, swap buffers */

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glLoadIdentity();
        gluLookAt(1.0,1.0,1.0,0.0,0.0,0.0,0.0,1.0,0.0);
        glTranslatef(0.0, 3.0, 0.0);
        glRotatef(theta[0], 1.0, 0.0, 0.0);
        glRotatef(theta[1], 0.0, 1.0, 0.0);
        glRotatef(theta[2], 0.0, 0.0, 1.0);
```

```
    glColorPointer(3,GL_FLOAT, 0, colors);
    glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndices);

        glutSwapBuffers();
}

void spinCube()
{
    /* Idle callback, spin cube 2 degrees about selected axis */

        theta[axis] += 2.0;
        if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
        glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{
    /* mouse callback, selects an axis about which to rotate */

        if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
        if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
        if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
}

void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-4.0, 4.0, -3.0 * (GLfloat) h / (GLfloat) w,
            5.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    else
        glOrtho(-4.0 * (GLfloat) w / (GLfloat) h,
            4.0 * (GLfloat) w / (GLfloat) h, -3.0, 5.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}

void main(int argc, char **argv)
{
    /* need both double buffering and z buffer */

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("colorcube");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutIdleFunc(spinCube);
    glutMouseFunc(mouse);
    glEnable(GL_DEPTH_TEST); /* Enable hidden--surface--removal */
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glColorPointer(3,GL_FLOAT, 0, colors);
    glClearColor(1.0,1.0,1.0,1.0);
    glColor3f(1.0,1.0,1.0);
    glutMainLoop();
}
```

Build and run this Project: Insert a screenshot of your output.
ANSWER:

B2. As you can see, the interface to the cube rotations is not very smooth. Control the cube with a virtual trackball.

Create a file trackball.c with the following content:

trackball.c

```
/* trackball.c */
/* Rotating cube demo with trackball*/

#include <math.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

#define bool int
#define false 0
#define true 1

#ifndef M_PI
#define M_PI 3.14159
#endif


int     winWidth, winHeight;

float   angle = 0.0, axis[3], trans[3];
bool    trackingMouse = false;
bool    redrawContinue = false;
bool   trackballMove = false;

/*----------------------------------------------------------------------*/
/*
** Draw the cube.
*/
GLfloat vertices[][3] = {
   {-1.0,-1.0,-1.0},{1.0,-1.0,-1.0}, {1.0,1.0,-1.0}, {-1.0,1.0,-1.0},
   {-1.0,-1.0,1.0}, {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}
};

GLfloat colors[][3] = {
    {0.0,0.0,0.0},{1.0,0.0,0.0}, {1.0,1.0,0.0}, {0.0,1.0,0.0},
    {0.0,0.0,1.0}, {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}
};


void polygon(int a, int b, int c , int d, int face)
{

    /* draw a polygon via list of vertices */

    glBegin(GL_POLYGON);
            glColor3fv(colors[a]);
            glVertex3fv(vertices[a]);
            glColor3fv(colors[b]);
            glVertex3fv(vertices[b]);
            glColor3fv(colors[c]);
            glVertex3fv(vertices[c]);
            glColor3fv(colors[d]);
            glVertex3fv(vertices[d]);
    glEnd();
}

void colorcube(void)
{

    /* map vertices to faces */

    polygon(1,0,3,2,0);
    polygon(3,7,6,2,1);
    polygon(7,3,0,4,2);
    polygon(2,6,5,1,3);
    polygon(4,5,6,7,4);
    polygon(5,4,0,1,5);
}

/*----------------------------------------------------------------------*/
/*
** These functions implement a simple trackball-like motion control.
*/

float lastPos[3] = {0.0F, 0.0F, 0.0F};
int curx, cury;
int startX, startY;

void
trackball_ptov(int x, int y, int width, int height, float v[3])
```

```
{
   float d, a;

   /* project x,y onto a hemi-sphere centered within width, height */
   v[0] = (2.0F*x - width) / width;
   v[1] = (height - 2.0F*y) / height;
   d = (float) sqrt(v[0]*v[0] + v[1]*v[1]);
   v[2] = (float) cos((M_PI/2.0F) * ((d < 1.0F) ? d : 1.0F));
   a = 1.0F / (float) sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
   v[0] *= a;
   v[1] *= a;
   v[2] *= a;
}


void
mouseMotion(int x, int y)
{
   float curPos[3], dx, dy, dz;

   trackball_ptov(x, y, winWidth, winHeight, curPos);
if(trackingMouse)
{
   dx = curPos[0] - lastPos[0];
   dy = curPos[1] - lastPos[1];
   dz = curPos[2] - lastPos[2];

   if (dx || dy || dz) {
         angle = 90.0F * sqrt(dx*dx + dy*dy + dz*dz);

         axis[0] = lastPos[1]*curPos[2] - lastPos[2]*curPos[1];
         axis[1] = lastPos[2]*curPos[0] - lastPos[0]*curPos[2];
         axis[2] = lastPos[0]*curPos[1] - lastPos[1]*curPos[0];

         lastPos[0] = curPos[0];
         lastPos[1] = curPos[1];
         lastPos[2] = curPos[2];
   }
}
   glutPostRedisplay();
}

void
startMotion(int x, int y)
{
    trackingMouse = true;
    redrawContinue = false;
    startX = x; startY = y;
    curx = x; cury = y;
    trackball_ptov(x, y, winWidth, winHeight, lastPos);
            trackballMove=true;
}

void
stopMotion(int x, int y)
{

    trackingMouse = false;

    if (startX != x || startY != y) {
            redrawContinue = true;
    } else {
            angle = 0.0F;
            redrawContinue = false;
            trackballMove = false;
    }
}

/*----------------------------------------------------------------*/

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

    /* view transform */

    if (trackballMove) {
                glRotatef(angle, axis[0], axis[1], axis[2]);

    }
            colorcube();

    glutSwapBuffers();
}

/*----------------------------------------------------------------*/

void mouseButton(int button, int state, int x, int y)
{
            if(button==GLUT_RIGHT_BUTTON) exit(0);
            if(button==GLUT_LEFT_BUTTON) switch(state)
```

```
        {
   case GLUT_DOWN:
        y=winHeight-y;
        startMotion( x,y);
        break;
   case GLUT_UP:
        stopMotion( x,y);
        break;
   }
}

void myReshape(int w, int h)
{
   glViewport(0, 0, w, h);
   winWidth = w;
   winHeight = h;
}

void spinCube()
{
   if (redrawContinue) glutPostRedisplay();
}



void
main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutCreateWindow("colorcube with trackball");
   glutReshapeFunc(myReshape);
   glutDisplayFunc(display);
   glutIdleFunc(spinCube);
   glutMouseFunc(mouseButton);
   glutMotionFunc(mouseMotion);
        glEnable(GL_DEPTH_TEST);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
        glOrtho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
   glMatrixMode(GL_MODELVIEW);
   glutMainLoop();
}
```

Please examine the way this virtual trackball was implemented. This gives a smoother interface.
Build and run this Project: Insert a screenshot of your output.
ANSWER:

B3. Paper and Pencil Explain GL_QUADS.

ANSWER: